ARL-TR-8474 ● AUG 2018

US Army Research Laboratory

Linear Layers and Partial Weight Reinitialization for Accelerating Neural Network Convergence

by John S Hyatt and Michael S Lee

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8474 ● Aug 2018

US Army Research Laboratory

Linear Layers and Partial Weight Reinitialization for Accelerating Neural Network Convergence

by John S Hyatt and Michael S Lee
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED (From - To)**
August 2018	Technical Report	May–August 2018

4. TITLE AND SUBTITLE
Linear Layers and Partial Weight Reinitialization for Accelerating Neural Network Convergence

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
John S Hyatt and Michael S Lee

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIH-C
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8474

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
We present two new approaches for accelerating the training of a neural network: 1) self-pruning using collapsible linear layers, and 2) mid-training weight reinitialization. By following each nonlinear layer with linear layers, then folding these linear layers into subsequent nonlinear layers after training, we are able to reproduce the benefits of overparameterizing the network, then pruning individual elements after training. We also periodically reinitialize the weights of nonlinear elements that do not improve the network's performance during training, freezing retained weights for several epochs to force the reinitialized weights to accommodate information already learned. Both methods demonstrate substantial gains: the resulting models are simpler than those attained by standard pruning and initialization methods, require fewer computations to train, and are more accurate than networks trained with those methods.

15. SUBJECT TERMS
machine learning, pruning, network compression, explainability, image painting

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Michael S Lee
Unclassified	Unclassified	Unclassified	UU	24	**19b. TELEPHONE NUMBER (Include area code)** (410) 278-5888

Contents

List of Figures

List of Tables

1. Introduction

Neural network research to date has prioritized the development of accurate, reliable models. However, advances have often come at the expense of ever-increasing network size and complexity,[1,2] with the result that many state-of-the-art machine learning models today are effectively black boxes.[3] Network simplicity is a critical consideration for developers and end users alike, for two reasons.

The first reason is efficiency. Neural networks have many applications in areas where time and resources (e.g., computing power, memory, power) are strictly limited. A cell phone cannot perform the same number of floating point operations per second (FLOPS) as a supercomputer or even a desktop, and a self-driving car has to make life-or-death decisions in a fraction of a second. Clearly, a slow, computationally costly, complex model will fail to satisfy in such cases, no matter how accurate or reliable.

The second reason is explainability. The more complex a model, the less likely a human can understand it. Which parts of the network are important, and which can be discarded? How could it be improved? Is this model a better fit for my problem, or is that one? Why did it make this decision? Can I trust it? These are questions that cannot be asked of a black box.

Increasing awareness of these problems has begun to move the field, leading to a resurgence of interest in network simplification methods like pruning,[4] the strategic removal of noncontributing network elements. Pruning, in combination with the appropriate data compression techniques, can dramatically reduce a network's size without compromising performance.[5]

However, only a fully-trained network can be pruned and compressed, so this method does nothing to expedite network training, which can be quite time-consuming and computationally expensive. Moreover, it is not always trivial to determine ahead of time whether a given element's contribution to the network is significant, and generally this determination is made on the basis of a somewhat arbitrary criterion.

These types of network-downscaling operations are necessary because, in practice, networks are almost always extremely overparameterized.[4,6] Overparameterization helps a neural network avoid becoming trapped in an unfavorable local minimum during training and failing to converge. The more weights a network has, the more weights are likely to be randomly initialized with values that favor convergence, and the more paths the gradient has available to approach the solution. As a result,

generally speaking, smaller networks require more epochs to converge than larger ones trained on the same data.[6]

We present two new methods for training a dense network that offer considerable improvements in accuracy and rate of convergence while reducing the network size before training:

- **Inclusion of collapsible linear layers.** The network is overpopulated with linear layers before training. A linear layer, which simply outputs linear combinations of the input, does not provide the network with increased capacity, since it can simply be folded into the weights of the subsequent layer. However, surprisingly, we find that it provides the same benefits as overparameterization (faster convergence and better accuracy) due to the increased number of parameters present during training. By collapsing the layers after training we remove these excess parameters automatically without reducing the amount of information contained in the network.
- **In-training partial weight reinitialization.** Rather than initializing a large number of parameters once, at the beginning of training, and then pruning the noncontributing ones at the end, we periodically reinitialize noncontributing parameters during training. To keep the network stable, we freeze the retained parameters for several epochs. We find that reinitializing in this manner allows a smaller network to substantially outperform a larger one, even if the number of reinitialized weights is smaller than the difference between the two networks' total parameter budgets.

2. A Simple Problem: Image Painting with Shallow Networks

We begin with a very simple dense network, shown in Fig. 1. It has two inputs, a single hidden layer containing three ReLU nodes, and an output layer containing one sigmoid node. Each (x, y) corresponds to a pixel in an image, and the network output is the value of that pixel.[7,8]

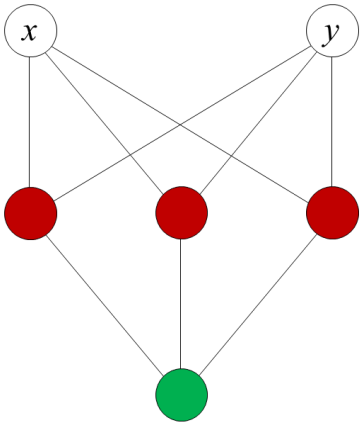

Fig. 1 A simple dense network with inputs *x* and *y*. Hidden ReLU nodes are represented in red and the output sigmoid node in green.

We train the network to reproduce images such as those in Fig. 2a–d. The pixels in these images have values of either 0 (purple) or 1 (yellow), and are arranged in shapes with six or fewer vertices. Thus, they can be described by a linear combination of three half-planes, corresponding to the three ReLU nodes in our network, passed through a sigmoid function.

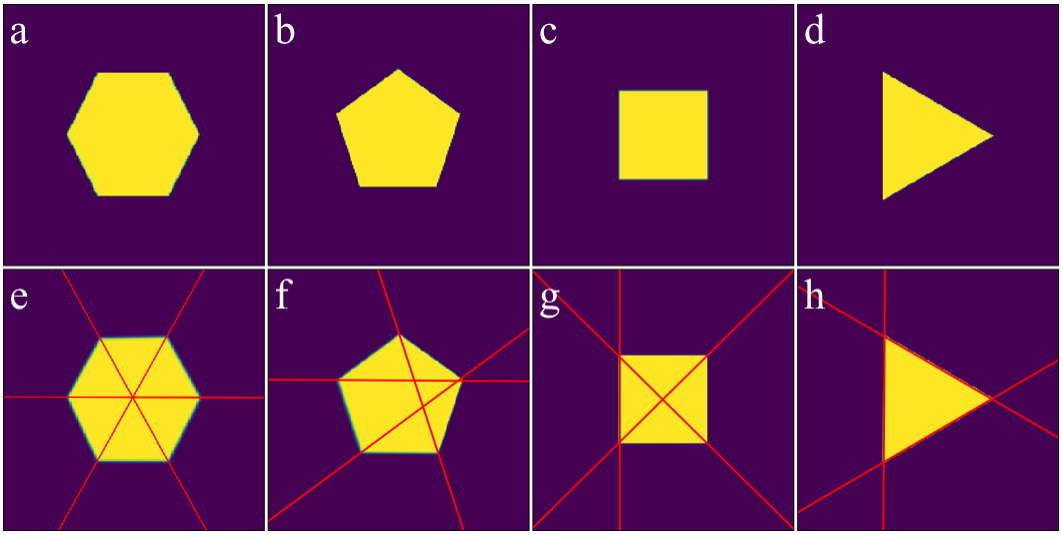

Fig. 2 The family of shapes that can be expressed by the network shown in Fig. 1 using only three ReLU nodes (the minimal solution). Panels a–d show the images used to train the network. Panels e–h show the images predicted by the network, with decision boundaries corresponding to the ReLU nodes shown in red.

A successfully trained network reproduces these simple images very accurately, as shown in Fig. 2e–h. The decision boundaries of the ReLU nodes are depicted in red. A breakdown of contributions from each node is shown for the case of the square in Fig. 3.

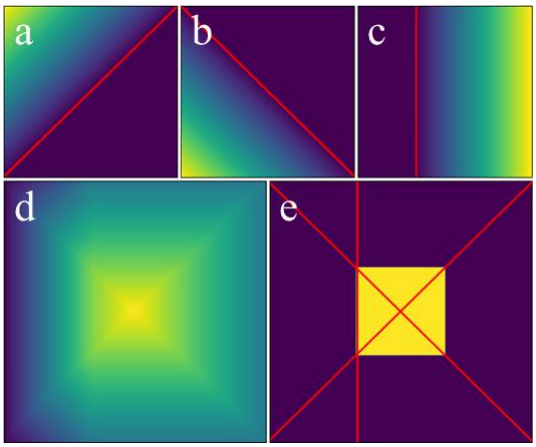

Fig. 3 The simplest possible solution to the square. Panels a–c show the outputs of the individual ReLU nodes, with the activation lines in red. Panel d shows the linear combination of the ReLU outputs that is fed into the output sigmoid node. Panel e shows the final result after the sigmoid, together with all three decision boundaries.

Note that, while we are evaluating the network using the same collection of (x, y) it trained on, we are only trying to encode the images into the network, and are not trying to generalize. In other words, the network's intended purpose is to memorize (overfit) the data.

Three further points merit discussion. First, each of the solutions shown in Fig. 2 is the simplest possible solution corresponding to that image, but each image can also be described with $n > 3$ ReLU nodes, or $N > 1$ hidden ReLU layers. In general, neural networks rarely produce the simplest solution; more often, they result in haphazard combinations of partial and/or redundant solutions that produce the same final image, but are not so easily understood. Such networks also almost always end up containing a large number of dead or otherwise noncontributing nodes. Finally, a larger network requires more memory to store and more calculations to evaluate, important considerations in real, more complex problems.

Second, training is not always successful. Depending on the random initialization of the weights before training, one or more of the ReLUs may be trapped in an unfavorable local minimum from which it cannot escape. This can either result in an unbounded shape (if there are fewer than three decision boundaries) or a bounded shape that is missing corners or otherwise distorted. These undesirable results are more common when fewer ReLU nodes are included in the network, since fewer nodes means fewer chances to get favorably initialized weights. The same limits can lead to excessively long training times for smaller networks.

Third, it is a rule of thumb that larger networks require more data to train than smaller ones.[9] It is therefore possible that smaller networks might be more robust to small data sets, if their other problems could be solved.

The problems discussed previously represent areas of active research as they also plague the more complex networks (e.g., deep convolutional networks) demanded by more complex problems. Networks train inefficiently if they are small, but express inefficient solutions if they are large, to say nothing of the resources large networks require.

Despite their simplicity, study of these networks provides helpful understanding, in part because the "right" answer is trivially obvious. This is definitely not the case with, for example, a convolutional network used in image recognition. In the following sections we use these networks to demonstrate that our new methods, *inclusion of collapsible linear layers* and *in-training partial weight reinitialization*, can reproduce the benefits of overparameterization without actually increasing the number of parameters in the network.

All of our work is done in Keras on top of a TensorFlow backend, but these methods could be implemented in any neural network programming environment.

3. Virtual Overparameterization with Collapsible Linear Layers

Linear layers have been used on several occasions to factorize weights in deep networks.[10–12] The idea is that an $m \times n$ weight matrix can be expressed as the product of two matrices with dimensions $m \times r$ and $r \times n$. For small enough r, the second representation has fewer parameters than the first.

We note that, in the context described previously, r acts as a bottleneck, decreasing the amount of information contained in a very wide, shallow network with many redundant parameters. Our variation on the method allows the training process to target a smaller network from the beginning and does not remove any information from the system.

3.1 Description of the Method

The dense network shown in Fig. 1 can be equivalently expressed with a matrix equation:

$$\begin{pmatrix} n_{11} \\ n_{12} \\ n_{13} \end{pmatrix} = \text{relu}\left[\boldsymbol{A}_1 \begin{pmatrix} x \\ y \end{pmatrix} + \boldsymbol{b}_1\right], \tag{1}$$

$$z = \text{sigmoid}\left[\boldsymbol{A}_2 \begin{pmatrix} n_{11} \\ n_{12} \\ n_{13} \end{pmatrix} + \boldsymbol{b}_2\right], \tag{2}$$

where (x, y) are the input coordinates, $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are weight tensors, $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are bias vectors, (n_{11}, n_{12}, n_{13}) are the outputs of the ReLU nodes in the hidden layer, and z is the pixel value predicted by the network. Because of the nonlinear activation functions present in these layers, the weight tensors and bias vectors cannot be combined using the algebraic distributive property. However, this is not the case for a linear layer.

Consider the networks shown in Fig. 4. Here, linear nodes are represented in gray. The matrix equation for the network shown in panel a is

$$\begin{pmatrix} n_{11} \\ n_{12} \\ n_{13} \end{pmatrix} = \text{relu}\left[\boldsymbol{A}_1 \begin{pmatrix} x \\ y \end{pmatrix} + \boldsymbol{b}_1\right], \tag{3}$$

$$\begin{pmatrix} n_{21} \\ n_{22} \\ n_{23} \\ \vdots \\ n_{29} \end{pmatrix} = \boldsymbol{A}_2 \begin{pmatrix} n_{11} \\ n_{12} \\ n_{13} \end{pmatrix} + \boldsymbol{b}_2, \tag{4}$$

$$z = \text{sigmoid}\left[\boldsymbol{A}_3 \begin{pmatrix} n_{21} \\ n_{22} \\ n_{23} \\ \vdots \\ n_{29} \end{pmatrix} + \boldsymbol{b}_3\right]. \tag{5}$$

However, in this case the distributive property allows us to fold Eq. 4 into Eq. 5:

$$z = \text{sigmoid}\left[\boldsymbol{A}_2{}' \begin{pmatrix} n_{11} \\ n_{12} \\ n_{13} \end{pmatrix} + \boldsymbol{b}_2{}'\right], \tag{6}$$

where $\boldsymbol{A}_2' = \boldsymbol{A}_3\boldsymbol{A}_2$ and $\boldsymbol{b}_2' = \boldsymbol{A}_3\boldsymbol{b}_2 + \boldsymbol{b}_3$. This process can be performed for any number of linear layers, regardless of their size, meaning that a network with linear layers contains exactly the same information as an otherwise identical network where those layers have been folded into the weights of subsequent nonlinear layers.

The key idea is that while these networks have identical capacity in principle, they do not have the same number of parameters. The model shown in Fig. 1 has 9 parameters (weights + biases), while those shown in Figs. 4a and 4b have 55 and

49, respectively. Further, their structures are different, meaning that they update differently during training as the loss error is backpropagated through the network.

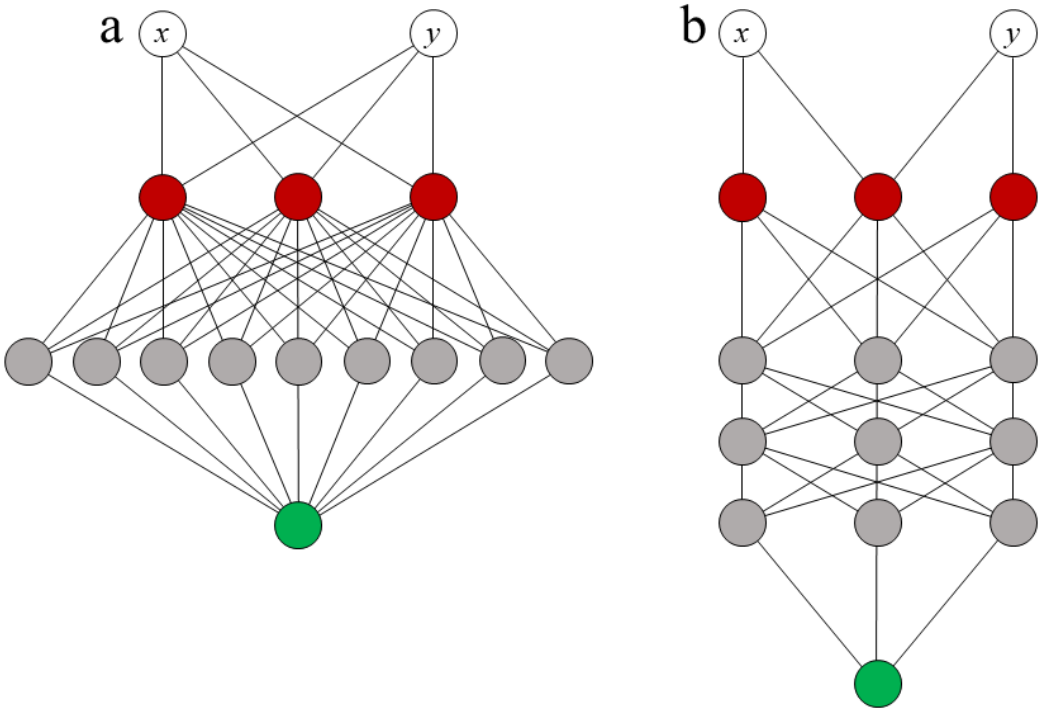

Fig. 4 Two example networks equivalent to the one depicted in Fig. 1. ReLU nodes are represented in red, linear nodes in gray, and the output sigmoid node in green.

3.2 Numerical Experiments

We compare several different models and find that a simple network containing linear layers dramatically outperforms an equivalent network without linear layers. However, surprisingly, networks with linear layers also, in some respects, outperform a network without linear layers, but with more ReLU nodes, subject to the same parameter budget.

We randomly initialize and train each model 30 times using the same protocol. Each iteration is given a practically unlimited number of epochs to train on (500,000), but we use Keras's built-in early stopping callback (patience 20) to stop when the minimum loss (encountered so far) has not decreased for 20 consecutive epochs. The batch size is 128. The loss metric is mean squared error. Before training, x and y are rescaled to lie between $[-0.5,0.5]$ and shuffled.

The model shown in Fig. 1 is denoted *model 1* and the models shown in Figs. 4a and 4b *model 2* and *model 3*, respectively. We also compare these to a larger shallow network with 12 ReLU nodes and no linear layers, denoted *model 4*.

Table 1 shows the results. ReLU nodes were considered "active" if their removal from the network, without changing any other weights, increased the loss by a factor of less than 1.5. In most cases, this corresponded to a node whose nonzero half-plane fell outside of the image area. Models 1–3 can only reproduce the square if all three nodes are active; model 4 has more options.

Table 1 Effect of collapsible linear layers

Model	# parameters	# active nodes	# minimal correct solutions	Loss (3 active nodes)[a]	Run time(s) (3 active nodes)[a]
1	9	1.8 ± 0.81	6	$(2.9 \pm 0.1) \times 10^{-3}$	640 ± 80
2	55	2.8 ± 0.5	26	$(2.7 \pm 1.3) \times 10^{-4}$	190 ± 40
3	49	2.4 ± 0.7	15	$(0.9 \pm 1.8) \times 10^{-4}$	100 ± 30
4	49	6.1 ± 1.3	1	2.9×10^{-3}	651

[a] Because only 1 of the 30 experiments with model 4 resulted in the minimal solution (3 active nodes), we cannot provide standard deviations for the loss and run time for this case.

Our results show that, of models with only three ReLU nodes, the ones with linear layers (models 2 and 3) were much more likely to train successfully than model 1. Moreover, they took substantially less time to reach a solution, and the loss was a factor of 10 lower, corresponding to visibly sharper edges on the predicted square.

Model 4 reproduced the square during all 30 trials, but only found the minimal solution one time. The other 29 trials resulted in overparameterized solutions with between 4 and 8 active nodes; detailed results from the trials of model 4 are shown in Table 2.

Table 2 Results from model 4 trials

# active nodes	# occurrences	Loss	Run time(s)
3	1	2.9×10^{-3}	651
4	3	$(7 \pm 9) \times 10^{-4}$	470 ± 110
5	4	$(1.4 \pm 0.5) \times 10^{-3}$	550 ± 130
6	11	$(1.3 \pm 0.2) \times 10^{-3}$	460 ± 50
7	7	$(6 \pm 3) \times 10^{-4}$	460 ± 90
8	4	$(2.8 \pm 1.2) \times 10^{-7}$	630 ± 30

Even though the number of parameters in model 4 is similar to those in models 2 and 3, model 4 was considerably slower to converge and (except for the solutions with eight active nodes) did not produce a significantly lower loss. We note that evaluating each ReLU node's loss to determine which nodes are active adds about 2 s to the run time per node, adding up to 10–20 s increased run time for model 4 relative to models 1–3. The rest of the time variation between models is due to models 1 and 4 requiring thousands more epochs to converge than models 2 and 3. Overall, this simple study implies that collapsible linear layers can reproduce at least some of the benefits seen in deeper networks without actually increasing the depth of the final network.

4. In-Training Weight Reinitialization

At the beginning of training, all weights and biases in the network are randomly initialized. During conventional training, this is only done once. However, later evaluation of the fully trained network shows that, typically, only a small fraction of the network contributes to the solution,[4,6] and the rest can be deleted with little effect on the network.

We find that, unsurprisingly, many noncontributing nodes are locked into a "bad" state early on in the training process. Rather than waiting until the end of training to prune these nodes, we give them a "fresh start" by reinitializing them during training. This produces higher-quality solutions at the cost of slightly increased training time and calculations required per attempt. To our knowledge, this type of training has not been performed before, although we are aware of one very recent paper that applied a similar concept to very sparse networks optimized for neuromorphic chips.[13]

4.1 Description of the Method

The procedure for training with reinitializations combines conventional training and pruning methods, outlined as follows and illustrated with a flowchart in Fig. 5:

1) Randomly initialize the model's weights and biases.
2) Train the model until the loss reaches a relatively stable value using Keras's built-in early stopping callback. In our experiments, we set patience to 20.
3) Evaluate the model on the training set, taking the resultant loss as a baseline.
4) Make a copy of the model.

5) Evaluate each ReLU node to determine whether it is contributing to the solution. We consider two categories of noncontributing nodes: low-loss nodes and trivially redundant nodes.

    a) **Low-loss nodes.** Nodes that can be deleted from the network without increasing its loss above a certain threshold, which we set at 1.5 times the baseline, are considered noncontributing.

    b) **Trivially redundant nodes.** Take the normalized dot product of the vectors formed by the weights (including bias) of each pair of nodes. We consider node pairs where this value is 0.99 or more to be trivially redundant, and consider larger groups of nodes trivially redundant as long as each node in the group is trivially redundant with at least one other.

6) Freeze all contributing nodes so that they cannot be updated during training. We accomplish this by applying a mask to their incoming weights, biases, and outgoing weights that sets their gradient to zero during the backpropagation phase of each epoch. Because this functionality is not built into Keras, we had to add a new function to the `Optimizers` base class in `optimizers.py`.

7) Reinitialize noncontributing nodes. These nodes are not frozen.

8) Train the partially frozen network for a few epochs to force the newly randomized nodes to accommodate information already learned by the network (i.e., the weights and biases of the contributing nodes). In our experiments, we set patience for this subtraining to 10.

9) Evaluate the model and compare the loss to the baseline. If the new loss is not comparable to or less than the baseline, revert the model to the copy made in step 4. We use 1.25 times the baseline as our loss threshold. This step is necessary because even with contributing nodes frozen, it is possible for the network to occasionally become trapped in a less favorable configuration than before reinitialization.

10) If the model loss has not significantly worsened as a result of reinitialization, unfreeze all nodes (remove the gradient mask).

11) Train the unfrozen network. We set patience to 20.

12) Repeat steps 4–11 as many times as desired.

13) After the last reinitialization and subsequent training, prune any remaining noncontributing nodes from the network. We use the same criteria from step 5.

14) Perform a final training of the network. We set patience to 20.

The flowchart does not include steps unrelated to reinitialization, such as collapsing linear layers.

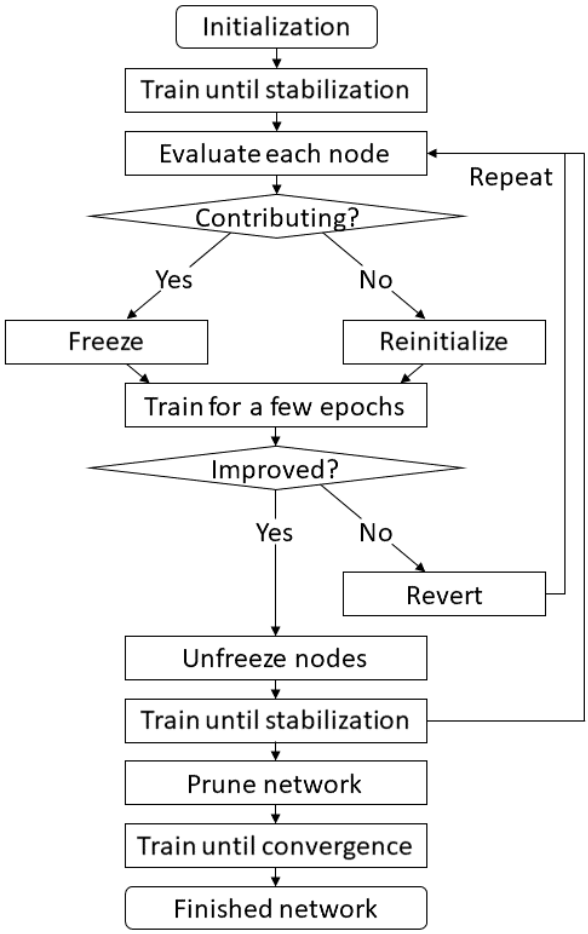

Fig. 5 Flowchart illustrating the reinitialization process

4.2 Numerical Experiments

We test three different combinations of numbers of ReLU nodes and numbers of reinitializations: 30 ReLU nodes and 0 reinitializations, 15 nodes and 1 reinitialization, and 10 nodes and 2 reinitializations. In each of these, the maximum number of weight initializations that can occur is 30, although because some weights are retained in the experiments with in-training reinitialization, in practice more weights are initialized in the first set of experiments than the others. We also

include three linear layers after each ReLU layer, each with the same size as the ReLU layer.

For our test image, we use a six-pointed star, shown in Fig. 6 along with a minimal solution that only requires six active ReLU nodes. This solution might be considered the "right" answer, as it is both easy to understand and predicts the image perfectly. Even for this comparatively simple image, however, a minimal solution is difficult to achieve, requiring many reinitializations of a simple model with six ReLU nodes. A more complex solution, with more active ReLU nodes, might more reliably reproduce the star, but would be more difficult to understand.

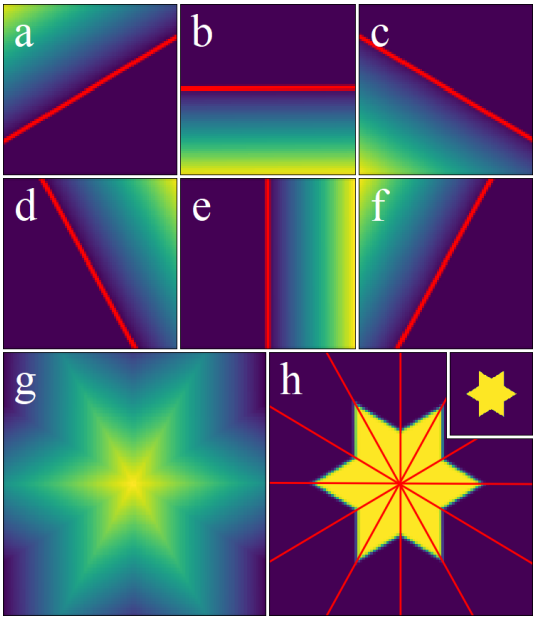

Fig. 6 A minimal solution for the star. Panels a–f show the outputs of the individual ReLU nodes, with activation lines in red. Panel g shows the linear combination of the ReLU outputs that is fed into the output sigmoid node. Panel h shows the final result after the sigmoid, together with all six decision boundaries. The inset in panel h shows the original image.

Table 3 shows the results of 30 experiments with each model. We find that even though the set with 30 ReLU nodes and 0 reinitializations has the largest total number of weight initializations, it performs substantially the worst.

Table 3 Effect of in-training weight reinitialization

# reinitializations / # nodes	# sides	# active nodes	# sides per active node	Loss	Run time(s)
0 / 30	7 ± 4	7 ± 4	0.8 ± 0.5	$(8 \pm 7) \times 10^{-2}$	90 ± 30
1 / 15	11 ± 2	10 ± 2	1.1 ± 0.2	$(2.3 \pm 1.5) \times 10^{-2}$	115 ± 10
2 / 10	10 ± 2	8 ± 1	1.3 ± 0.2	$(2.7 \pm 1.5) \times 10^{-2}$	114 ± 13

First, it mostly produces output shapes that are dramatically different from the star, as measured in the "# sides" column, containing the average and standard deviation of the number of sides of the figure produced at the end of training. Examples of such images are shown in Fig. 7. As a result, this set also produces output with the largest loss.

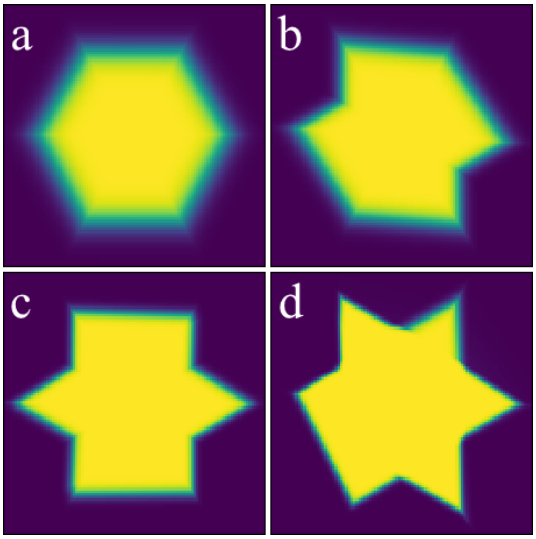

Fig. 7 Example images produced by failed networks with a) 6, b) 8, c) 10, and d) 11 sides

In contrast, the sets with one and two reinitializations are more likely to produce the correct answer, or one that is at least close. This is borne out further in Fig. 8, which compares the number of images produced with a given number of sides across all three sets of experiments. Only the largest network, trained without any reinitializations, ever failed completely: in 7 out of 30 instances it produced a blank image (0 sides), and it reproduced the star only 3 times.

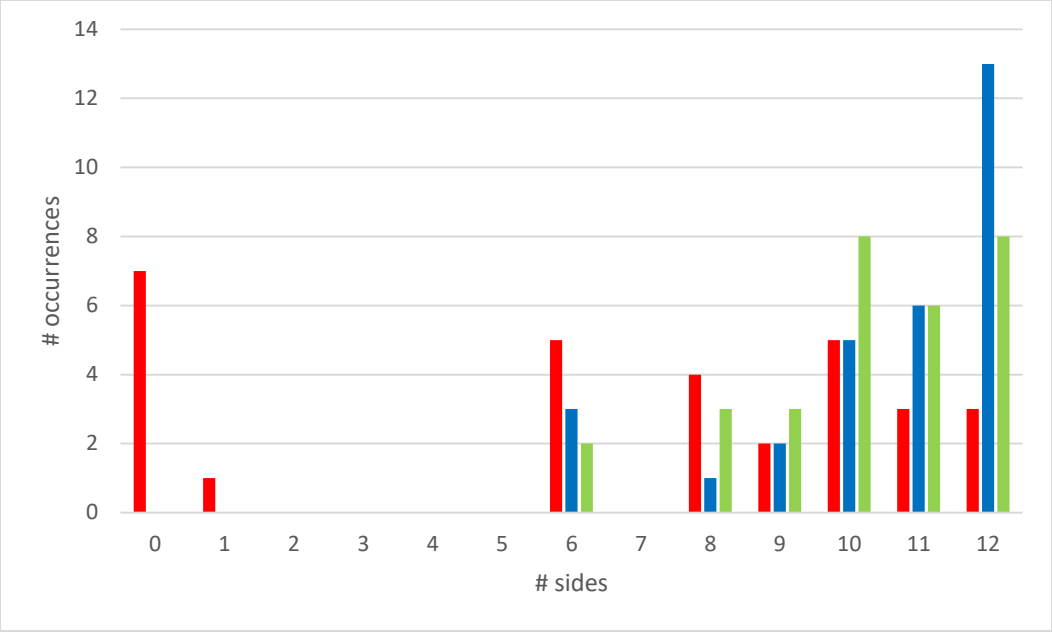

Fig. 8 Comparisons between networks trained with differing numbers of nodes and numbers of reinitializations: 30 ReLU nodes and 0 reinitializations (red), 15 and 1 (blue), and 10 and 2 (green). Without reinitialization, even the largest network often fails to converge to any solution.

Less obviously, the non-reinitialized network is also less efficient in the way it produces its results. The ratio of the number of sides in an output image to the number of active nodes required to produce that output gives an indicator of the nodes' independence and expressive power. (A similar argument could be made about the product of the loss and the number of active nodes.) In the minimal answer depicted in Fig. 6, this ratio would be 2. Even with reinitializations, none of our experiments achieve that goal, but they are able to increase the ratio above 1, which is not the case for the non-reinitialized network.

We finally note that all three sets of experiments have comparable runtimes; although reinitialization increases the time to train by about 25%, it decreases the average final loss by more than double that amount. At least for these experiments, the tradeoff is worth it.

5. Conclusions and Further Work: Application to CNNs

We have shown that including collapsible linear layers and reinitializing weights during training both significantly improve the outcome of training simple, fully connected neural networks. To the best of our knowledge, both of these methods are new.

As a result of our focus on a bottom-up approach to explainability, this work focuses on extremely simple, shallow networks, which limits the scope of our conclusions. However, these new methods can be straightforwardly extended to the more application-relevant problem of convolutional neural networks (CNNs):

- **Inclusion of collapsible linear layers.** While a linear layer cannot be inserted directly between convolutional layers, the same effect can be achieved with a set of $1 \times 1 \times n$ convolution kernels with linear activations, which can be collapsed in similar fashion to a linear fully connected layer.
- **In-training partial weight reinitialization.** The procedure would not change except that kernels, not nodes, would be reinitialized.
- **Trivially redundant kernels.** While the method for identifying trivially redundant nodes as described in section 4.1 does not generalize well to large fully connected networks due to the size of the weight vectors, we expect it to prove more useful in a CNN. Because convolution kernels are fairly small (3×3, 5×5, etc.) even in large networks, trivially redundant kernels can likely be identified by a simple dot product, rather than with a more complicated procedure (e.g., comparing their outputs).
- **Circularly permuted kernels.** The output of a convolution kernel is approximately equal to the output produced by a second kernel whose rows and columns are circular permutations of those in the first kernel, provided the input is large compared to the kernel. This is frequently the case, at least in the shallower layers of the network. This implies that, under such circumstances, a kernel is also trivially redundant with all of its circular permutations.

Considering the computational expense associated with training CNNs, any improvement due to these methods is likely to have a considerable impact. We are in the process of updating our code to apply to CNNs to test this assertion.

6. References

1. Goodfellow I, Benjio Y, Courville A. Deep learning. Cambridge (MA): MIT Press; 2016 [accessed 2018 Mar 15]. https://www.deeplearningbook.org.

2. Szegedy C, Liu W, Yangqing J, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. Paper presented at: CVPR 28. 2015 IEEE Conference on Computer Vision and Pattern Recognition; 2015 Jun 7–12; Boston, MA.

3. Gilpin LH, Bau D, Yuan BZ, Bajwa A, Specter M, Kagal L. Explaining explanations: an approach to evaluating interpretability of machine learning. arXiv preprint 2018;arXiv:1806.00069.

4. LeCun Y, Denker JS, Solla SA. Optimal brain damage. In: Touretzky DS, editor. Advances in Neural Information Processing Systems 2. NIPS 2; 1989 Nov 27–30; Denver, United States. San Francisco (CA): Morgan Kaufmann Publishers; 1990.

5. Han S, Mao H, Dally WJ. Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. Paper presented at: ICLR 2016. Proceedings of the 6th International Conference on Learning Representations; 2016 May 2–4; San Juan, Puerto Rico.

6. Frankle J, Carbin M. The lottery ticket hypothesis: finding small, trainable neural networks. arXiv preprint 2018;arXiv:1083.03635.

7. Pascanu R, Montúfar G, Bengio Y. On the number of response regions of deep feedforward networks with piecewise linear activations. arXiv preprint 2013;arXiv:1312.6098.

8. Karpathy A. ConvnetJS demo: image painting. Stanford (CA): Stanford University; c2014-2018 [accessed 2018 Aug 6]. https://cs.stanford.edu/people/karpathy/convnetjs/demo/image_regression.html.

9. Figueroa RL, Zheng-Treitler Q, Kandula S, Ngo LH. Predicting sample size required for classification performance. BMC Med Inform Decis Mak. 2012;12(8). doi: 10.1186/1472-6947-12-8.

10. Ba J, Caruana R. Do deep nets really need to be deep? In: Ghahramani Z, Welling M, Cortes C, Lawrence ND, Weinberger KQ, editors. Advances in Neural Information Processing Systems 27. NIPS 27; 2014 Dec 8–11; Montréal, Canada. Red Hook (NY): Curran Associates, Inc; 2014.

11. Sainath TN, Kingsbury B, Sindhwani V, Arisoy E, Ramabhadran B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. Paper presented at: ICASSP 38. 2013 IEEE International Conference on Acoustics, Speech, and Signal Processing; 2013 May 26–31; Vancouver, Canada.

12. Xue J, Li J, Gong Y. Restructuring of deep neural network acoustic models with singular value decomposition. In: Bimbot F, editor. 14th Annual Conference of the International Speech Communication Association. INTERSPEECH 14; 2013 Aug 25–29; Lyon, France. Red Hook (NY): Curran Associates, Inc; 2013.

13. Bellec G, Kappel D, Maass W, Legenstein R. Deep rewiring: training very sparse deep networks. Paper presented at: ICLR 2018. Proceedings of the 6th International Conference on Learning Representations; 2018 Apr 30–May 3; Vancouver, Canada.

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
9 (PDF)	RDRL CIH S D SHIRES M VINDIOLA RDRL CIH C E CHIN M LEE J HYATT S EDWARDS P WANG RDRL SLE W M MARKOWSKI A BEVEC